Synthesis of Heterogeneous Pipelined Multiprocessor Systems using ILP : JPEG Case Study

Haris Javaid Sri Parameswaran
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{harisj, sridevan}@cse.unsw.edu.au

ABSTRACT

Streaming applications can be implemented with a pipeline of processors. Each processor in the pipeline can be an Application Specific Instruction Set Processor (ASIP) with the result being a heterogeneous pipelined MPSoC system. Since ASIPs can be of differing configurations, finding the optimal set of configurations for a multiprocessor architecture is a difficult problem.

In this paper, we obtain an optimal system design for a set of processors which execute a multimedia application. The variables in the system are the presence or absence of different additional instructions and differing cache configurations for each of the processors. The problem is formulated as a *0-1 Integer Linear Programming (ILP)* problem. To reduce the complexity of the *ILP* formulation, inferior ASIP configurations are efficiently pruned so that the solution could be reached quickly. Given a system runtime constraint, the proposed methodology finds a design with minimal area. We integrated this design methodology into a commercial design flow, and performed a case study upon the JPEG encoding application. We obtained 15 optimal designs subject to 15 different runtime constraints, each in less than 100 seconds from more than 4.2×10^{13} design points.

Categories and Subject Descriptors

C.1.3 [**Other Architectural Styles**]: Heterogeneous (hybrid) Systems, Pipeline Processors; C.4 [**Performance of Systems**]: Modeling Techniques, Design Studies

General Terms

Algorithms, Design, Performance

Keywords

MPSoCs, Design Space Exploration, Integer Linear Programming

1. INTRODUCTION

Increased transistor density in modern chips has enabled multi-core chips in embedded devices. MPSoCs facilitate parallel hardware with partitioned software to attain performance gains and decrease power consumption. MPSoCs are becoming a viable solution as it becomes more difficult to increase clock speeds, and to continuously widen instruction level parallelism.

MPSoCs can be either homogeneous or heterogeneous. Homogeneous MPSoCs use identical processing entities, where as Heterogeneous MPSoCs contain different processing elements. These processing elements can include coprocessors, general purpose processors, and application specific architectures such as ASICs or DSPs. Heterogeneity in MPSoCs allow the designed systems to occupy smaller area footprint and consume less power by matching each of the processing elements with the task assigned to them. To improve efficiency while still allowing the flexibility of a processor, Application Specific Instruction-set Processors (ASIPs) [1, 2, 4] could be used as the basic building blocks of MPSoCs. An ASIP's architecture and instruction set can be matched to the needs of a specific application, improving performance while minimizing area. ASIPs in an MPSoC environment exploit both coarse- and fine-grained parallelism available within an application.

Heterogeneous pipelined multiprocessor architecture is a viable implementation platform for streaming applications. In a pipelined architecture, processing entities are connected in a pipelined structure via queues. Each pipeline stage may contain one or more processing elements which process the incoming data stream. The data stream goes through each stage until the output is finally written by the last pipeline stage. Pipelined architecture provides better performance gains for streaming applications, as each of its stage can be customized to suit a particular part of the application which is executed by that pipeline stage processor(s) [17].

A JPEG encoder is described in this paper, which is implemented with a multiprocessor system configured in a pipelined fashion. The JPEG application is manually partitioned into standalone processes to be executed on individual processors. Since ASIPs can be configured in various ways, the problem of mapping a partitioned application onto available ASIP configurations is examined. The designer specifies the runtime of the pipelined system, and the proposed design methodology searches for the optimal design point which meets that runtime constraint while minimizing the area.

The rest of the paper is organized as follows: Section 2 provides an overview of the work already done in the MPSoC domain and Section 3 specifies the details of the model used for the pipelined multiprocessor system. Section 4 explains the overall design methodology to implement heterogeneous MPSoCs using ASIPs while Section 5 states the formulation of the mapping problem as a *0-1 ILP* problem. Our algorithm to prune the design space is discussed in Section 6. Section 7 provides the experimental setup to validate the proposed methodology with the results presented in Section 8. Finally, Section 9 summarizes the paper.

2. RELATED WORK

Many multimedia applications have already been implemented onto multiprocessor architectures. These include a real time video and graphics management system [19] and an HDTV system [8]. Different techniques in multiprocessor architectures have been explored to speed up applications. Kodaka et al. [13] used both coarse- and fine-grained parallelism in a multiprocessor chip called OSCAR. OSCAR exploited task level parallelism, loop pipelining and instruction level parallelism simultaneously. The work in [12] partitioned loops into several pipeline stages and used an iterative algorithm to exploit greater parallelism and reduced area in the designed system. Banerjee et al. [7] used a Signal Flow Graph (SFG) to partition an application

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-470-6/08/10 ...$5.00.

to be implemented as a heterogeneous multiprocessor system. The authors targeted scheduling of pipelined tasks on multiple processors, showing that macro pipelining based scheduling increases the throughput rate to several times that of homogeneous multiprocessor scheduling techniques.

Optimization of heterogeneous multiprocessor architectures has also been widely explored. Different heuristics have been proposed to efficiently explore the design space of a heterogeneous multiprocessor system. Sun et al. [20] examined multi-ASIP systems, integrating custom instruction selection for ASIPs with assignment and scheduling of tasks. An iterative algorithm was used to perform these two steps simultaneously and find a multiprocessor system with minimum runtime given an area budget for the custom instructions. Cong et al. [9] explored partitioning of applications represented as cyclic directed graphs onto a multiprocessor architecture with processors connected in a pipeline. The approach used graph algorithms such as labelling, clustering and packing to generate a multiprocessor system with minimized latency and number of processors under throughput constraints. The authors in [18] targeted implementation of streaming applications onto heterogeneous pipelined multiprocessor systems using ASIPs as processors. The work presented a methodology to rapidly explore the design space of such a multiprocessor architecture. The authors used *runtime* $\times$ *area* as cost function to be minimized for the overall system design, while exploring different ASIP configurations.

Integer Linear Programming (ILP) [10] is a widely used technique in optimization of multiprocessor architectures. Schwiegershausen et al. [15] presented a Mixed Integer Linear Programming (MILP) based approach for optimization of heterogeneous multiprocessor systems, but the presented approach becomes impractical as the number of processor types increase. The authors in [11] also formulated the synthesis of customized multiprocessor systems as an MILP problem considering pipelined execution of task graph onto available processors. Kuang et al. [14] presented an ILP based approach for integrated hardware/software partitioning and pipelined scheduling of partitioned tasks onto heterogeneous multiprocessor architecture.

The present work enhances current design methodologies for multiprocessor architectures by utilizing *Integer Linear Programming* technique to synthesize an optimal heterogeneous pipelined multiprocessor system. Our work differs from all of the above in the following ways:

1. Design space exploration of a heterogeneous pipelined multiprocessor system (implementing a streaming application) is formulated as a *0-1 ILP* problem. To the best of our knowledge, this work is the first to address design space exploration as a *0-1 ILP* problem in the context of heterogeneous pipelined multiprocessor systems.
2. This work also combines strategic pruning of the design space to rapidly and optimally solve the problem.

3. SYSTEM MODEL

This case study is aimed at semi-automated implementation of a streaming application onto a heterogeneous pipelined multiprocessor system. Processing entities in the multiprocessor system are connected in a pipelined fashion to exploit task-level parallelism available in these applications [17]. Each pipeline stage of the system contains an ASIP to execute some part of the targeted application. The target application is partitioned into processes to be mapped onto these ASIPs/pipeline stages. Input data is read by the first pipeline stage and is processed by later stages until it is finally written out by the last pipeline stage. Each ASIP can be customized to generate different processor configurations, which form the design space of this case study. We focused on the rapid exploration of the generated design space and obtained optimal designs (for differing system runtime constraints) using a *0-1 ILP* solver for the pipelined system.

3.1 Application

We performed our case study on a freeware JPEG encoding algorithm implemented in C. The JPEG algorithm divides an image into

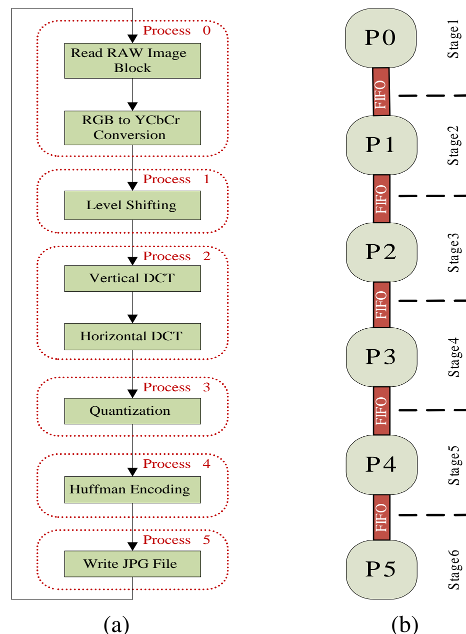


Figure 1: (a) Major operations of JPEG algorithm as partitioned into processes (b) Six processors configured in a pipeline with one processor in each stage of pipeline.

macro blocks which are then processed one by one. The application is manually partitioned into processes that can be mapped onto processors/pipeline stages as standalone programs. It is first profiled over a single processor system. Then, profiling information is used to partition the application into processes to approximately balance the workload of each process. In other words, input block rate of the first process should be the same as output block rate of the last process. Processes communicate with each other using queues (FIFOs) which allow communication at a much higher bandwidth, devoid of the contention of a typical shared bus architecture. Figure 1(a) shows the major steps of a JPEG encoding algorithm which are partitioned into a set of connected processes while Figure 1(b) implements a six processor pipeline system to execute the mapped processes. The first processor reads the raw image block and performs RGB to YCbCr conversion. The second processor is assigned the level shifting process. Similarly, other processors are allocated different tasks as shown in Figure 1. Readers interested in further details are referred to [17]. Since input images to the pipelined system are processed at the macro block level, time to process one macro block by a processor is referred as latency of that processor. As can be seen, while one processor is processing one macro block other processors will be busy processing other macro blocks in a typical pipelined fashion, resulting in performance gains.

3.2 Processor Configurations

Each processor in the multiprocessor implementation can be configured in one of the configurations available for that processor. Processor configurations differ by the additional instructions they contain and by the sizes of their instruction and data caches. Since processes have already been mapped onto processors in the pipeline, each processor is customized according to the process being executed on it. A commercial ASIP design tool from Tensilica Inc. [4] is used to automatically generate these configurations from a base processor for each of the processors in the pipelined system. We used the same base processor for each stage of the pipeline (however, base processors can be different for different pipeline stages and can be specified by the designer). Table 1 depicts the details of the base processor which was used in our experiments.

The processor configuration generation can also be controlled by the designer using the *overhead granularity* value. One set of addi-

Speed	563 MHz
Pipeline Depth	5
Core Area	112048 gates
Instruction Width	8 bytes
PIF Interface	128 bytes
Load/Store Units	1
Other Features	MAC16 & MUL32 instructions, Boolean Registers, Direct associativity of instruction & data cache with 16 bytes line size

Table 1: Base Processor Configuration

tional instructions for a processor can be different from another set of additional instructions for the same processor if their area overhead differs by more than the specified *overhead granularity*. We used a value of 5000 (gates) as *overhead granularity* and only changed instruction and data cache sizes from 1KB to 32 KB to obtain customized configurations of each processor. Area of a processor configuration includes base processor, additional instructions (if any), and instruction and data cache sizes. Table 2 shows the number of configurations for each processor that were generated for the six-processor pipelined system shown in Figure 1(b). DCT and Huffman Encoding are computationally-intensive operations of the JPEG encoder which will generally increase the number of additional instructions, thus increasing the number of configurations for these two processors. The total design space is a permutation of all these configurations.

3.3 Runtime Calculation

Shee et al. [18] used a runtime estimation technique to rapidly explore the design space of a pipelined heterogeneous system. The estimation technique uses an equation to calculate pipelined system runtime with one set of processor configurations as:

$$\mathbb{R} = \mathrm{R}^{init}(p_0) + \sum_{i=0}^{N-1} \mathrm{L}(p_i) + (I-1) \times \mathrm{L}(p_{critical}) + \mathrm{R}^{final}(p_{N-1}) \quad (1)$$

L refers to latency of a processor in a pipeline stage and R^{init}, R^{final}, I and N refer to initialization time of first processor, finalization time of last processor, number of macro blocks and number of processors respectively. Using this equation, system runtime can be estimated using latencies of individual processors instead of simulating the whole system with that particular set of processor configurations. Processor latencies include inter-processor communication delays, but exclude read and write FIFO stalls. This means only net communication time and processing time of one macro block is used. As slower processors stall for the critical processor, FIFO stalls should not be a part of the calculation of system runtime. We used Equation 2 which is same as equation 1 but with several modifications.

$$\mathbb{R} = \mathrm{R}^{init}(p_0) + \sum_{i=0}^{N-1} \mathrm{L}^1(p_i) + (I-1) \times \mathrm{L}(p_{critical}) + \mathrm{R}^{final}(p_{N-1}) \quad (2)$$

$$\text{where} \qquad \mathrm{L}(p_i) = \sum_{j=2}^{I} \mathrm{L}^j(p_i)/(I-1)$$

The reason for separating first latency L^1 from averaged latency L (averaged over I-1 macro blocks except the first one) of a processor corresponds to the fact that there will be more instruction or data cache misses while processing the first macro block. To make runtime estimation more accurate, first latencies are recorded separately and used for those processors which are non-critical. For the critical processor, averaged latency is used because once the pipeline is

Pipeline Stage Processor	Number of Configurations
Processor 0	144
Processor 1	144
Processor 2	396
Processor 3	144
Processor 4	252
Processor 5	144

Table 2: Number of configurations generated for each processor

filled, that is, every processor has processed first macro block then next macro blocks will be output by the last pipeline stage after every L clock cycles, which is the average latency of the critical pipeline stage processor. Equation 2 can be used to estimate the system runtime given one set of processor configurations.

To rapidly explore the whole design space, instead of simulating every possible combination of processor configurations to get system runtime, each processor's configurations are simulated separately [16]. This is possible because computation time of a processor is separated from FIFO stalls and only net communication time is recorded. A 3-tuple number (L^1, L, A) is recorded for each processor configuration where A stands for area of that particular configuration. A 4-tuple number (R^{init}/R^{final}, L^1, L, A) is recorded to include initialization/finalization time for each configuration of first and last processor. The pipelined system will be simulated for K_{max} times which is the number of configurations for the processor having maximum configurations from amongst all the processors as other processor's configurations are simulated at least once. System runtime for every possible combination of processor configurations can be calculated using Equation 2. As shown in Equation 3, area of the pipelined system is the summation of the area cost of all processors where area cost of each processor is measured in gates.

$$\mathbb{A} = \sum_{i=0}^{N-1} A_i \quad (3)$$

Area of the pipelined system does not include area cost of FIFOs that connect processors as the application is partitioned only once at the start. The designer knows how much data (number of bytes) will be transferred between two processors and can specify FIFO sizes and their associated area costs. Since it will be a constant value, there is no need to account for FIFO areas. In this work, we did not consider the cost of processor memories.

4. OVERALL DESIGN METHODOLOGY

The overall design flow for implementation of pipelined heterogeneous multiprocessor systems is as follows: A pipeline of N processors and partitioned processes of an application are given. The processors are numbered P_0, P_1 to P_{N-1}. Processor P_i has a number of configurations numbered $P_{i,0}, P_{i,1}$ to P_{i,K_i-1} where K_i is the total number of configurations for processor P_i. These configurations are generated from the provided base processor, which is then enhanced by additional instructions. The additional instructions vary by greater than a given fixed value, *overhead granularity*, which is specified in number of gates. For example, if *overhead granularity = 5000 gates*, then size of processor $P_{i,1}$ >= $P_{i,0}$ + *5000* excluding the size of caches. Each configuration $P_{i,j}$ is associated with a tuple as described in Section 3.3. For example, $P_{0,0}$ will be associated with $(R^{init}_{0,0}, L^1_{0,0}, L_{0,0}, A_{0,0})$. The design space consisting of processor configurations is pruned efficiently as described in Section 6 by removing all inferior configurations subject to R_d, where R_d is the given runtime constraint. One configuration of each processor is selected so that the resulting pipelined system has minimum area while its runtime is less than R_d. The problem of mapping processes onto processor configurations, and finding the best set of processor configurations from the pruned design space is formulated as a *0-1 ILP* problem, and solved using an ILP Solver which provides an optimal solution. Pruning the design space for different values of R_d and finding an optimal solution using an ILP Solver will reveal differ-

ent optimal design points with respect to R_d. These optimal design points (which are obtained for multiple given runtime constraints) are referred to as pseudo Pareto optimal points of the whole design space, and are a truncated set of Pareto optimal points.

5. ILP FORMULATION

The mapping problem can be stated as follows: *Given an application partitioned into processes and processors with their configurations, processes are mapped to processor configurations such that system area is minimized while system runtime is less than designer's runtime*. In the ILP formulation of the mapping problem, N denotes the number of processors while K_i denotes the total number of configurations for processor i. $L^1_{i,j}$ and $L_{i,j}$ refers to L^1 and L of processor i in configuration j respectively as explained before. The different steps for ILP formulation of the mapping problem are:

5.1 Decision Variables

Basic decision variables are introduced to reflect mapping of a process onto a processor configuration specified as:

1. $x_{i,j}$ *equals 1* if configuration j of processor i is selected
2. $c_{m,n}$ *equals 1* if processor m with configuration n is selected as the critical processor of the pipelined system

The necessity for $c_{i,j}$ decision variables arises from the fact that there can only be one critical processor which has the maximum averaged latency amongst the selected processor configurations. This condition is formulated as a constraint described in section 5.3.

5.2 Objective Function

The objective function is to minimize area of the multiprocessor system given the system runtime is within the upper bound specified by the designer which reflects the soft realtime constraint of streaming applications. The objective function is stated below.

$$\text{Minimize} \sum_{i=0}^{N-1} \sum_{j=0}^{K_i-1} A_{i,j} x_{i,j}$$

where $A_{i,j}$ stands for area of the processor i with configuration j.

5.3 Constraints

The different constraints applicable to the pipelined multiprocessor system are listed below:

1. For each processor exactly one configuration can be selected.

$$\sum_{j=0}^{K_i-1} x_{i,j} = 1 \qquad \forall\, i$$

2. Only one processor can be selected as critical pipeline stage processor.

$$\sum_{i=0}^{N-1} \sum_{j=0}^{K_i-1} c_{i,j} = 1$$

3. A processor configuration already selected can be selected as critical processor configuration.

$$c_{i,j} - x_{i,j} \le 0 \qquad \forall\, i,j$$

4. System runtime must be less than or equal to designer's runtime.

$$\sum_{j=0}^{K_0-1} R^{init}_{0,j} x_{0,j} + (I-1) \times \sum_{i=0}^{N-1} \sum_{j=0}^{K_i-1} L_{i,j} c_{i,j} + \sum_{i=0}^{N-1} \sum_{j=0}^{K_i-1} L^1_{i,j} x_{i,j} + \sum_{j=0}^{K_{N-1}-1} R^{final}_{N-1,j} x_{N-1,j} \le R_d$$

where R_d refers to system runtime constraint.

5. A processor configuration is critical only if its averaged latency L is maximum amongst the averaged latencies of all selected processor configurations.

$$Max(L_{1,j}) \times [1 - c_0] + \sum_{j=0}^{K_0-1} L_{0,j} c_{0,j} \ \ge\ \sum_{j=0}^{K_1-1} L_{1,j} x_{1,j}$$

where $Max(L_{1,j})$ stands for maximum L amongst all configurations of processor 1 and c_0 is summation of all $c_{0,j}$ for processor 0. If processor 0 is selected as the critical processor then c_0 will be 1, zeroing the factor $Max(L_{1,j}) \times [1 - c_0]$ and allowing the comparison that critical latency is greater than the latency of processor 1. In other words, if processor 0 is non-critical then the factor $\sum_{j=0}^{K_0-1} L_{0,j} c_{0,j}$ dies down and the above constraint is always evaluated as true. There will be *N-1* constraints for each processor as it has to be compared against every other processor. Thus, in total we have *N(N-1)* such constraints which ensure that critical processor is actually the one with maximum averaged latency amongst the selected processor configurations.

6. PRUNING DESIGN SPACE

The *0-1 ILP* formulation results in $2 \times \sum_{i=0}^{N-1} K_i$ decision variables which can become large very quickly. Reducing the design space directly reduces the number of decision variables, which will help the ILP solver to find the optimal design more quickly. The design space is pruned to reduce the number of configurations of each processor. The basic idea behind design space pruning arises from the runtime constraint imposed on the final design. This constraint leads to the fact that there will be some configurations of a processor that cannot be part of the optimal system design and could be removed from the design space. If the runtime constraint is greater than the highest runtime supported by the design space, not even a single configuration is removed. Similarly, for a runtime constraint less than the lowest possible runtime supported, all the configurations are pruned resulting in an empty design space.

Algorithm 1 depicts our approach to prune the design space. The algorithm selects one processor P_i, and goes through all of its configurations while other processors are set to those configurations with minimum execution times. Three different times namely *ProcessingTime*, *CriticalProcessingTime* and *Latency* are defined as follows:

$$\textit{Latency} = L \quad \textit{for all processors}$$

$$\textit{ProcessingTime} = \begin{cases} R^{init} + L^1 & \text{if } \textit{first} \text{ processor} \\ R^{final} + L^1 & \text{if } \textit{last} \text{ processor} \\ L^1 & \textit{other} \text{ processors} \end{cases}$$

$$\textit{CriticalProcessingTime} = \begin{cases} R^{init} + L^1 + (I-1) \times L & \text{if } \textit{first} \text{ processor} \\ R^{final} + L^1 + (I-1) \times L & \text{if } \textit{last} \text{ processor} \\ L^1 + (I-1) \times L & \textit{other} \text{ processors} \end{cases}$$

MinProcessingTime(P_i), *MinCriticalTime(P_i)* and *MinLatency(P_i)* functions return configuration of processor P_i having minimum *ProcessingTime*, *CriticalProcessingTime* and *Latency* respectively. It should be noted that a configuration $P_{0,x}$ of P_0 having minimum *Latency* might not have minimum *ProcessingTime* as well. Three arrays *min_proc_times*, *min_latencies* and *min_crit_times* are used to store the processor configurations according to the above criteria (lines 5-9). For example, *min_proc_times[0]* corresponds to a configuration $P_{0,j}$ which had minimum *ProcessingTime* amongst all configurations of processor P_0. Similarly, *min_crit_times[0]* corresponds to a configuration $P_{0,i}$ which had minimum *CriticalProcessingTime* amongst all configurations of processor P_0.

The algorithm is illustrated here. Assume that processor 3 is selected as the critical processor having the worst *min_latency* (line 11). The algorithm goes through processor 1 configurations: all other processors at this instant are set to configurations from *min_proc_times*

Algorithm 1 Algorithm to prune design space

1: **Input:** $P_0, P_1, ...P_{N-1}$ where P_x array contains tuples associated with each configuration of processor P_x, and runtime constraint R_d
2:
3: **Output:** Inferior configurations in $P_0, P_1, ...P_{N-1}$ are removed
4:
5: **for** *i=0* to *N-1* **do**
6: *min_proc_times[i]* = MinProcessingTime(P_i);
7: *min_crit_times[i]* = MinCriticalTime(P_i);
8: *min_latencies[i]* = MinLatency(P_i);
9: **end for**
10:
11: *critical = Max(min_latencies)* where critical refers to processor number
12:
13: $L_{critical,cc}$ = $\mathbb{L}$(*min_crit_times[critical]*) where $\mathbb{L}$ returns value of L for a given configuration
14:
15: // Goes through every processor
16: **for** *i=0* to *N-1* **do**
17: Initialize *runtime_initial = 0;*
18: **for** *j=0* to *N-1* **do**
19: **if** *j != i* and *j != critical* **then**
20: *runtime_initial += min_proc_times[j];*
21: **end if**
22: **end for**
23: // Goes through processor P_i's all configurations
24: **for** *j=0* to K_i*-1* **do**
25: Initialize *runtime = runtime_initial;*
26: **if** *i == critical* **then**
27: Use $L_{i,j}$ as critical latency to calculate *runtime*
28: **else**
29: **if** $L_{i,j} > L_{critical,cc}$ **then**
30: Use $L_{i,j}$ as critical latency to calculate *runtime*
31: **else**
32: Use $L_{critical,cc}$ as critical latency to calculate *runtime*
33: **end if**
34: **end if**
35: **if** *runtime* $> R_d$ **then**
36: Delete configuration $P_{i,j}$
37: **end if**
38: **end for**
39: **end for**

(line 20) while the critical processor is set to configuration from *min_crit_times* (line 13). Any configuration of processor 1 that results in runtime greater than R_d is excluded from the design space because all other processor configurations have minimum possible times. There exists no possibility that the current configuration of processor 1 may have a combination with other processor configurations resulting in runtime less than R_d – this configuration of processor 1 cannot be a candidate for optimal system design and is deleted. The proposed algorithm also looks for the possibility that a processor other than the one having worst *min_latency* can be the critical processor in the final optimal design (lines 29-33), as opposed to heuristic developed in [18] where the critical processor is selected once at the start only. The design space is efficiently pruned, thus reducing the complexity of the ILP, but all the good configurations of each processor (with respect to R_d) are retained. It should be noted that the pruning algorithm directly depends on R_d so, the percentage reduction in design space and the number of decision variables for ILP formulation depends on the specified value of R_d. As the algorithm goes through each processor's configurations only once, its complexity is $O(N \times K_{max})$ where K_{max} is the number of configurations for the processor having maximum configurations from amongst all the processors.

7. EXPERIMENTAL SETUP

We used the Xtensa LX family of processors and Xtensa RB-2007.1 toolset from Tensilica Inc.[4] to design our multiprocessor system. This toolset includes a C/C++ compiler that compiles C/C++ code specifically for the targeted processor such as the one shown in Table 1. Tensilica's Instruction Set Simulator (ISS) is used to simulate LX processors while Xtensa Modeling Protocol (XTMP) is used to describe the multiprocessor system. XTMP provides an environment to instantiate multiple processors and connect them to FIFOs, rapidly setting up and simulating multiprocessor systems. XTMP uses ISS to simulate the processors which can produce profiling data such as total clock cycles, global stalls etc.

Queues were used to model communication between processors where queues are simple FIFOs implemented in C. FIFO interface includes *push* and *pop* functions that are used by the connected processors to write to and read from a FIFO respectively. The Queue interface provided by LX processors is used to connect each processor to a FIFO in the XTMP environment. Stalling logic is automatically created for a processor with a queue interface, so a *push* to a full FIFO or a *pop* from an empty FIFO results in a processor being stalled. Clock cycles spent while a processor is stalled due to the queue interface are recorded as global stalls by ISS which can then be used to calculate net execution time of a processor.

The RB-2007.1 toolset also includes Xtensa PRocessor Extension Synthesis (XPRES), which generates customized processor directly from the C/C++ code executed by the base processor. Analyzing the C code, XPRES automatically generates new instructions and register files specified in TIE (Tensilica Instruction Extension) language. The newly generated instructions may consist of any combination of fused operations, FLIX instructions [5], vector operations and specialized operations [6]. XPRES is used to create tailored processor configurations using the *overhead granularity* parameter by automatically generating multiple TIE files reflecting different customizations of the base processor. The generated TIE files are readily compiled through the TIE compiler and the source code does not need to be modified to take advantage of customized versions of the base processor.

We used *lp_solve* [3], a free software, to solve the formulated *0-1 ILP* problem. *lp_solve* reads an input file in LP format specifying the *ILP* problem and outputs a text file specifying the values of decision variables. Perl scripts generate processor configurations using Tensilica's ASIP tool and simulates them to record the tuples as described in section 3.3. Other Perl scripts prune the design space, write input file for *lp_solve* and read the generated output file to retrieve the actual processor configurations. Time spent for these steps of the design methodology is recorded to obtain the total time for the design space exploration.

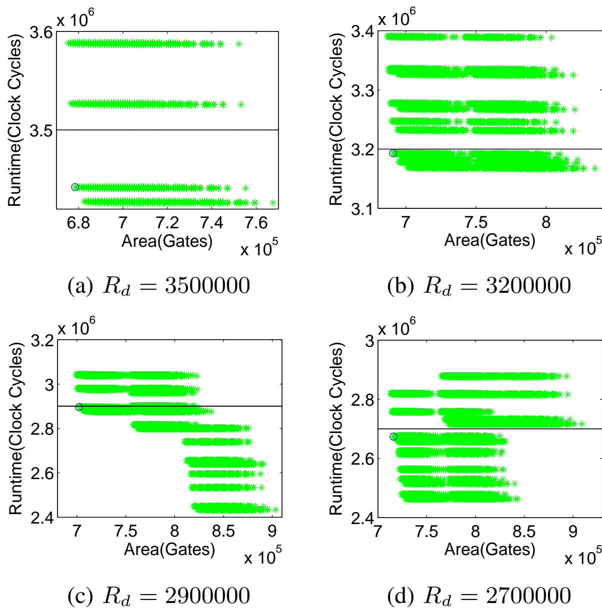
8. RESULTS & ANALYSIS

Figure 2 shows the design space exploration of the JPEG encoder as implemented in this case study. The figures reflect only a small part of the whole design space consisting of more than 4.2×10^{13} system designs, impractical to be plotted. Figure 2(a) used 3,500,000 clock cycles as designer's runtime constraint. As can be seen, a major portion of the design space will be pruned by the proposed algorithm helping the ILP solver to reach the optimal system design quickly. Illustrated by the black line in each of the figures is the runtime constraint that was given. Blue circle in the figures show the optimal design selected by the ILP solver. Figure 2(d) shows that there are points even above the optimal result specifying a runtime more closer to 2,700,000, but those designs have a greater footprint and are not selected by the ILP solver.

Table 3 shows the actual runtimes and areas of the system designs obtained from ILP solver for different values of R_d. As can be seen in column 4, optimal designs had different critical processors for different values of R_d. This signifies the fact that any processor can be critical in the final optimal design.

The designer can prune the design space and use ILP solver again and again specifying different runtimes to obtain the pseudo Pareto optimal points of the design space where processor configurations are simulated only once. Figure 3 shows pseudo Pareto optimal points for R_d ranging from 3,700,000 to 2,500,000 in steps of 100,000 clock cycles which covers the whole design space. The last two pseudo pareto points were obtained using 2,740,000 and 2,440,000 values for R_d because there are no design points below 2,400,000.

Generation of processor configurations depicted in Table 2 took 8 minutes, while 19 hours were spent in simulation of the gener-



(a) $R_d = 3500000$ (b) $R_d = 3200000$

(c) $R_d = 2900000$ (d) $R_d = 2700000$

Figure 2: Part of whole design space with different runtime constraints provided by the designer

R_d (Clock Cycles)	Actual Runtime (Clock Cycles)	Actual Area (Gates)	Critical Processor
2700000	2673799	715938	Processor3
2900000	2896105	702065	Processor4
3200000	3193054	690843	Processor4
3500000	3442314	678432	Processor2

Table 3: Runtime and Area of optimal designs from ILP Solver

ated processor configurations on a quad core machine running at 2.15 GHz. It took 3 minutes on average to simulate one set of processor configurations. Thus, if we go through the whole design space consisting of 4.2×10^{13} designs it would take years to get the simulation results. Time spent in obtaining optimal results using *lp_solve* are summarized in Table 4. These timings also include the time for writing the input file for *lp_solve* and reading generated output file to retrieve the actual configuration of each processor. The advantage of

R_d (Clock Cycles)	Full Search	Pruned Search
3500000	3 sec	2 sec
3200000	17 min	96 sec
2900000	35 min	4 sec
2800000	no sol. (2 days)	3 sec
2700000	no sol. (2 days)	1 sec

Table 4: Time comparison of ILP solver

pruning the design space using the proposed algorithm is the reduced space that the ILP solver has to search for an optimal solution. Reducing designer's runtime prunes a larger design space and ILP solver finds the optimal design more quickly as depicted in row 5 and 6 of Table 4. More importantly, pruning only removes configurations that violate R_d constraint, ensuring the final design found by ILP solver will actually be optimal.

9. CONCLUSION

In conclusion, we formulated the mapping problem of processes of a partitioned application onto ASIP configurations forming a multiprocessor system where ASIPs are connected in a pipeline. We showed that solving the mapping problem as a *0-1 ILP* problem with a design space pruning algorithm reveals each of the pseudo Pareto optimal designs in less than 100 seconds from a very large design space consisting of more than 4.2×10^{13} system designs. We implemented the whole design methodology in Perl and using commercial ASIP design tool and a multiprocessor environment. We will extend our

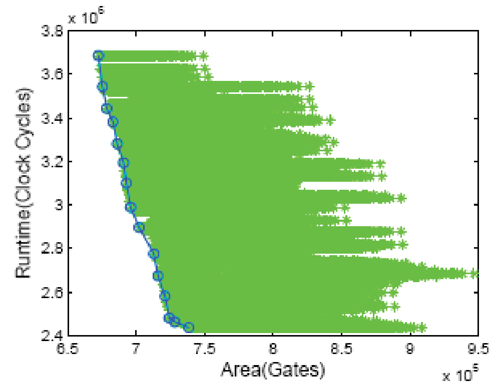


Figure 3: Pseudo Pareto optimal points of the design space with R_d ranging from 3,700,000 to 2,400,000

work by developing new heuristics that can search the design space faster, comparing their effectiveness with the results obtained using the presented methodology.

10. REFERENCES

[1] Altera Nios Processor. Altera Corp. (http://www.altera.com).

[2] ARC the leader in configurable processor technology. ARC International (http://www.arc.com).

[3] lp_solve. Available at: http://lpsolve.sourceforge.net/5.5/.

[4] Xtensa Processor. Tensilica Inc. (http://www.tensilica.com).

[5] Flix: Fast relief for performance-hungry embedded applications, 2005. Available at: http://www.tensilica.com/pdf/FLIX_White_Paper_v2.pdf.

[6] XPRES Generated Specialized Operations, 2005. Available at: http://tensilica.com/pdf/XPRES%201205.pdf.

[7] S. Banerjee, T. Hamada, P. Chau, and R. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *Signal Processing, IEEE Transactions on*, 43(6):1468–1484, 1995.

[8] A. Beric, R. Sethuraman, C. Pinto, H. Peters, G. Veldman, P. van de Haar, and M. Duranton. Heterogeneous multiprocessor for high definition video. *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 401–402, 7-11 Jan. 2006.

[9] J. Cong, G. Han, and W. Jiang. Synthesis of an application-specific soft multiprocessor system. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 99–107, New York, NY, USA, 2007. ACM.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stien. *Introduction to Algorithms*. MIT Press and MCGraw-Hill, Second edition, 2001.

[11] J. DeSouza-Batista and A. Parker. Optimal synthesis of application specific heterogeneous pipelined multiprocessors. *Application Specific Array Processors, 1994. Proceedings., International Conference on*, pages 99–110, 22-24 Aug 1994.

[12] J. Jeon and K. Choi. Loop pipelining in hardware-software partitioning. In *Asia and South Pacific Design Automation Conference*, pages 361–366, 1998.

[13] T. Kodaka, K. Kimura, and H. Kasahara. Multigrain parallel processing for jpeg encoding on a single chip multiprocessor. In *IWIA '02: Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, page 57, Washington, DC, USA, 2002. IEEE Computer Society.

[14] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao. Partitioning and pipelined scheduling of embedded system using integer linear programming. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05)*, pages 37–41, Washington, DC, USA, 2005. IEEE Computer Society.

[15] M. Schwiegershausen and P. Pirsch. A formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes. In *EURO-DAC '95/EURO-VHDL '95: Proceedings of the conference on European design automation*, pages 8–13, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[16] S. L. Shee. ADAPT: Architectural and Design Exploration for Application Specific Instruction-set Processor Technologies. PhD thesis, School of CSE, University of New South Wales, Sydney, September 2007.

[17] S. L. Shee, A. Erdos, and S. Parameswaran. Heterogeneous multiprocessor implementations for jpeg:: a case study. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 217–222, New York, NY, USA, 2006. ACM.

[18] S. L. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 811–816, New York, NY, USA, 2007. ACM.

[19] M. Strik, A. Timmer, J. van Meerbergen, and G.-J. van Rootselaar. Heterogeneous multiprocessor for the management of real-time video and graphics streams. *Solid-State Circuits, IEEE Journal of*, 35(11):1722–1731, Nov 2000.

[20] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSID '05: Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 551–556, Washington, DC, USA, 2005. IEEE Computer Society.